Analysis of Memory Use for Improved Design and Compile-time Allocation of Local Memory

By Geoffrey D. McNiven and Edward S. Davidson
Coordinated Science Laboratory
University of Illinois
Urbana IL 61801

Abstract

New trace analysis techniques are used to study memory referencing behavior for the purpose of designing new local memories and determining how to allocate them for data and instructions. In an attempt to assess the inherent behavior of the source code, the trace analysis system described here reduces the effects of the compiler and host architecture on the trace by using a technique called *flattening*. The variables in the trace, their associated single-assignment values, and references are histogrammed on the basis of various parameters describing memory referencing behavior. Bounds are developed specifying the amount of memory space required to store all *live* values in a particular histogram class. The reduction achieved in main memory traffic by allocating local memory is specified for each class.

1. Introduction

The traffic between a processor and main memory has become more of a bottleneck as processor operations have become faster relative to main memory accessing, and as more processors share the same main memory. The first problem is particularly evident in single-chip processors where the on-chip processing rate can be much faster than the rate at which data can be transferred off chip. Large mainframes and supercomputers also often have a memory bottleneck.

An appropriate solution to this traffic problem involves placing a small memory near the processor in a location where traffic to this local memory will have little adverse effect on performance. The idea of local memory is not a new one: both a cache and a register file qualify as such. Because the local memory is small, it must be used efficiently. Each location must hold a datum that will not only be referenced again, but referenced again soon, thus effecting the maximum reduction in traffic.

Future knowledge is generally unavailable to a cache. Thus, the cache controller is unable to allocate space based upon the future referencing patterns of data in the cache. Rather, allocation must be performed on the basis of past behavior, as in the case of LRU, FIFO, and LFU algorithms. During compilation, however, the compiler is able to acquire a rough future knowledge of memory referencing patterns. This knowledge cannot be exact in programs where the memory locations referenced are data dependent, as is the case in most programs. Using available future knowledge, a good compiler is able to make effective use of a register file.

Using a register allocation technique for cache allocation would result in a cache that is part of the architecture and receives directives from the compiler or the assembly language programmer. Instead of performing a detailed analysis of referencing behavior on the entire program, as would be done when performing register allocation, the compiler could recognize classes of variables that exhibit similar referencing behavior and allocate memory to a variable based upon its class membership.

The study reported here was performed with such a scheme in mind. Program traces are generated, and the variables in the trace are classified on the basis of statistics related to memory referencing behavior. These classes of variables exhibit similar memory referencing behavior, at least to the extent that their similar statistics imply. Furthermore, these statistics can be used to compute bounds on the amount of memory required to hold all live members of a class as well as bounds on the effect of the class on main-memory traffic. In many cases, class membership can be inferred by a compiler based on a variable's use in the source program. Thus, this analysis offers a method by which the compiler can assist the cache controller in allocating local memory. This method is far less costly than performing a complete register analysis on the program.

A question arises regarding which statistics on the referencing behavior of a variable indicate its contribution to traffic and memory use. Several statistics provide such an indication, including average interreference time, interreference time standard deviation, number of references, lifetime, number of deaths, average death time, and death time standard deviation. We classify the

variables in a program by these statistics, then use information regarding the size of the classes, and bounds derived from the class statistics to draw conclusions regarding design and allocation of local memory.

There is a wealth of work involving analysis of memory referencing behavior, cache performance, replacement algorithms, and register allocation. An excellent Survey of cache memories can be found in.[Smit82] A variety of cache measurement studies are available.[Stre83, Clar83, Haik84, AKCB86, PeSh77] The issues involved in cache measurement and workload choice are discussed in.[Smit85] The impact of cache on system performance is mentioned discussed in.[MGST70] A survey of replacement algorithms is available in.[Bela66] and an algorithm to minimize traffic is found in.[HKMW66] Register allocation algorithms are described in.[Day70, Beat74, Chai82]

The work described in this paper, however, is directed toward reducing local-to-main memory traffic instead of cache misses. While no replacement algorithm is specified, guidelines for developing a replacement algorithm are described, along with guidelines for designing local memory. Much of the philosophy behind our memory trace analysis system is unusual.

In the remainder of this paper, section 2 describes our memory analysis system, including the tracer, preprocessors, and the programs that analyze the trace and produce appropriate statistics. Section 3 offers some bounds on memory use based on the limits of the histogram classes and the number of values in each class. Section 4 presents the data collected from several traces and discusses their implications for design of local memory.

2. Tracing System

Our memory analysis system generates and analyzes traces under Berkeley UNIX on a VAX. Most trace analysis experiments are performed on simple address traces. Because clues to memory referencing information are available in the program structure, the traces described here include such information as opcodes, addressing modes, the limits of the text and initialized data spaces, and the values of the frame and argument pointers at procedure calls. Including this information

allows a broader analysis, including partitioning the variables according to the memory region in which they reside, such as stack, heap, text segment, etc.

The results of most machine-level address traces are, by nature, very dependent upon the compiler-architecture system on which they were collected. This dependence is important for studies related to a specific system; however, the design of new architectures should not be unduly biased by system-dependent aspects of measurements. Instead, a compiler and architecture independent trace is desirable. The tracing system described here attempts to reduce the dependence of the trace on the host architecture. The primary vehicle for achieving independence is a trace-processing program called the *flattener*.

Most general-purpose computer architectures include a multilevel memory hierarchy, usually consisting of a register set and a main memory. In addition, many systems utilize further, architecturally-invisible levels such as cache and, in virtual memory systems, physical memory and disk. Because this research involves allocation of local memory, analyzing a trace with register allocation already performed affects the results. The primary effects of register allocation that are visible in the trace are the use of registers to hold operands and addresses, and load and store instructions for loading registers, storing results, and performing spilling.

Compilers also affect the trace through the quality of the generated code. Poor generated code results in more instructions being executed when the program is run. In particular, poor compilers tend to generate extra move instructions. The move instructions are often used to store data in temporary locations, resulting in extra variables and references appearing in the trace. Furthermore, executing a move instruction creates an alias: if a datum is copied to another location without modification, then two different memory locations contain the same piece of data. Correspondingly, a single piece of data has two names: the addresses of these two memory locations.

The flattener reduces the dependence of the trace on the underlying compiler and machine architecture by removing aliases, move instructions, and the multilevel memory hierarchy. It also

maps addresses from the space of the memory hierarchy into a single level address space called the value space, containing values. A *value* is much like a variable; however, a value is a single-assignment entity: it is written just once, but may be read any number of times. The single-assignment property is essential if aliases are to be removed. If a variable is written several times, then it corresponds to several values, one for each write. A value becomes *live* when it is written—or read on the first reference to an input variable—and *dead* when it is read for the last time. A variable is live when one of its values is live and is dead otherwise.

Both memory locations and registers are mapped into the value space. Value names are assigned sequentially, starting with 1, as needed. All items in memory, including instructions, are considered to be values. Values are all considered to have the same size, regardless of their content or the sizes of their associated variables.

When the flattener processes the trace file, it associates a value name with each memory location in the trace. Whenever a read reference to a memory location is encountered, that reference is transferred to the flattened trace, with the associated value name in place of the memory address. Whenever a write reference to a memory location is encountered, the memory location is assigned a new value name, and the reference is transferred to the trace file with the new value name replacing the address of the memory location. A modify reference is treated as a read reference followed by a write reference. When a move instruction is encountered, no references are transferred to the trace file, instead the destination operand is associated with the same value name as the source operand. Treating move instructions in this fashion removes aliases. The move instructions are not included in the flattened trace file, because they are useless.

Performing flattening on the trace file reduces the effects of compiler-performed register allocation by renaming register variables and memory locations as values and eliding move instructions, while linking both the source and target of the move to the same value. Eliminating move instructions also removes some of the effects of poor compilation. A further result is that analyzing a flattened trace file allows us to gather data about memory referencing patterns of

variables only during their live periods. These periods are of interest to the memory designer, because if the allocation of live variables can be performed properly, then memory performance can be optimized.

Flattening is an optional process and is not appropriate for all analyses. A trace should be flattened when information concerning values is desired, and should not be flattened when information concerning variables is desired. For example, it is not appropriate to use a flattened trace to drive a cache simulator. The simulated cache performance would be artificially bad because in a flattened trace, different values of the same variable all have different names. If they all had the same name, as is the case in an unflattened trace, then as new values were created by writes to the variable, dead values would automatically be removed from the cache as they were overwritten by the new values.

Thus, flattening removes some effects of the underlying compiler-architecture system on which the trace was generated. Analyzing a flattened trace allows measurement of variable referencing behavior only during the periods when the variable is live. Live variable analysis removes the bias to the statistics caused by including a variable's referencing activity—*e.g.*, moves or unreferenced intervals—while it contains dead data.

The tracing system consists of the tracer, the flattener, a combination cache simulator and live variable analyzer, and a variety of programs for producing histograms, graphs and tables concerning memory use.

In any empirical study of performance, the question of the choice of workload always arises. We tried to choose programs that would provide interesting results for studying memory referencing behavior. Our tracing facility does not allow tracing of the operating system. The data presented here is from a few user programs that we have analyzed to date. The programs under consideration were chosen for their diversity, their representation of a certain type of workload, and their size. Our memory analysis system is unable to consider traces longer than about a million instructions, due to hardware limitations. Due to cpu usage constraints, the traces are one

hundred thousand instructions in length.

Because these traces do not cover the entire execution of most programs under consideration, the section of each program which was felt to be most representative of the behavior of the entire program was selected. For each large program, a profile was generated using gprof. Most of the programs displayed a small group of subroutines that account for the majority of the runtime. Tracing was initiated at the entrance to this group of popular subroutines, and continued for 100,000 consecutive instructions.

3. Bounds on Memory Space Requirements for Histogram Classes

To determine an upper bound for the memory requirements for a class of values, consider the two dimensional histogram of number of references and lifetime. The average interreference time, I, for a value is given by:

$$I = \frac{L}{(R-1)}$$

where L is the value lifetime, and R is the number of references to the value. How many values from particular histogram classes can be live at one time, *i.e.*, how much memory space might be required to hold them all?

First consider a class of values that is referenced exactly twice, with lifetimes from L_1 to L_2. To maximize the number of simultaneously live values from this class, let $L = L_2$, let the memory be initially empty at time $t = 0$, and assume that only values from the class under consideration are referenced. At time $t = 1$, a value may be referenced for the first time. This may be repeated until $t = L$. At this point, L values in this class are live. At $t = L + 1$, the value referenced at $t = 1$ will have been live for time L, and must be referenced for its second and final time. This must also be true for the next $L - 1$ references, at which time all L of these values must have received their final reference. No other reference pattern can produce more than L live values from this class. Thus, the maximum number of simultaneously live values from that class is L.

Now consider the general case of a histogram class of values that has lifetimes from L_1 to L_2 and are referenced R_1 to R_2 times. The maximum number of simultaneously live values in this class is reached if all values have the longest lifetimes, L_2, and the fewest references, R_1. Accordingly, let $L = L_2$ and $R = R_1$. Suppose that the maximum number of simultaneously live values is V and suppose they are all live at $t = L$. Then all V values are referenced first in the interval $[0, L)$ and last in the interval $[L, 2L)$. Thus, they are all referenced R times in the interval $[0, 2L)$, *i.e.*, $R \cdot V \leqslant 2L$. Thus,

$$V \leqslant \frac{2L}{R}$$

Note that if $R = 2$, then $V \leqslant L$ as above. One reference pattern that achieves $V = \frac{2L}{R}$ is as follows. Starting at $t = 0$, reference $\frac{L}{R}$ values $R - 1$ times each, then reference $\frac{L}{R}$ other values once each. Now $\frac{2L}{R}$ values are live and

$$t = \frac{L}{R}(R - 1) + \frac{L}{R} = L.$$

Beginning at $t = L$, reference the first $\frac{L}{R}$ values once each, for the last time, and then each of the second $\frac{L}{R}$ values for $R - 1$ consecutive times. Now all $\frac{2L}{R}$ values are dead and $t = 2L$. Note that the first and last values referenced had lifetime L and all others had lifetimes $\leqslant L$. Thus, the maximum number of simultaneously live values in the histogram class with lifetimes L_1 to L_2 and number of references R_1 to R_2 is

$$V = \frac{2L_2}{R_1}.$$

The argument above assumes that all of the references during the critical interval are to values from the class under consideration. During execution of most programs, this will not be the case. Instead, references from the other classes will be intermingled. The actual number of live values from each class will actually be significantly less than the bound, in most cases.

A metric of related interest is the average number of live values with lifetimes from L_1 to L_2. Bounds on the average can also be computed. Consider all the value lifetimes laid end to end in one long strip. The length of the strip is measured in references, and each value takes a length of strip equal to its lifetime in references. This strip is then chopped up into smaller segments, each of $S - r$ references, where S is the length of the program trace in references, and $0 \leq r < L$. A value lifetime cannot be cut in the middle, so the strips may not be equal to S, and $S - r$ is always equal to an integral number of lifetimes. The number of S-length segments is the average number of live values, V_{avg}, which can be bounded above and below by

$$\frac{N}{\left\lceil \frac{S}{L_1} \right\rceil} \leq V_{avg} \leq \frac{N}{\left\lceil \frac{S}{L_2} \right\rceil}$$

where N is the total number of values in this class and S is the length of the trace. The value lifetime L is chosen as L_1 for the lower bound and L_2 for the upper bound. An estimate of the average may be computed by using the average of L_1 and L_2 for L. The bounds on V_{avg} differ from those on the maximum V in that they use not only the parameters of the histogram classes, but also the results of the analysis which produces the number of values, N, in each class. Computing bounds for some of the classes and comparing them to actual counts of live values demonstrates the usefulness of these bounds as estimators, as shown in the next section.

The sum of the maximum memory requirements for all classes may be greater than the available local memory. In this case, an assessment is needed of the traffic caused by excluding certain values from the local memory. Consider a histogram in which the metric for each class is the fraction of total references. The fraction of references corresponds to the fraction of traffic, if no local memory is used. Thus, the fraction of references for a class indicates the fraction by which the total main memory traffic would be reduced if that class were put in local memory. On a more detailed level, one over the fraction of traffic indicates the average number of references between two references to the same histogram class.

4. Analysis of Trace Data

The trace analysis centers on a trace of the program troff, a text formatter running under UNIX. In addition we consider three other programs in less detail. These are *gauss*, *diff*, and *flt*. Gauss is a standard gaussian elimination program operating on a 10x10 matrix. Diff is a UNIX utility program that compares two files and prints the differences. Flt is the flattener described earlier. While data was being generated for this paper, flt became one of the major cpu users on the system and thus qualified for a spot in this study. All of the traces are 100000 instructions long.

Table 4.1 lists the programs and the number of references, values, and variables in each. The values are present in the flattened trace, while the variables are present in the unflattened trace. On the average, each trace has 5-9 values per variable. Considering that many of the values are instructions which are never written, this implies well over 5-9 writes per data variable. There are an average of 6-8 references per value.

Figure 4.1 shows a scatter plot of average interreference time vs. interreference time standard deviation for troff. Each dot represents a value. The plot shows 1000 points. Plotting more points than this adds little information because most of the points cluster near the origin. This graph shows three major classes of values. The first class has small interreference time and small standard deviation. This corresponds to variables that are referenced frequently and regularly during their lifetimes. This class is a likely candidate for allocation to memory because of the small interreference time.

Table 4.1. References, values, and variables per trace.

Program	References	Values	Variables
diff	254020	43807	4799
flt	236473	34121	8223
gauss	242870	31029	5562
troff	289517	35203	5042

The second class contains those values with a larger interreference time and a small standard deviation. This class contains values that are referenced infrequently, but regularly. Values in this class would not be good candidates for allocation in local memory if space is limited. While these values may be referenced repeatedly, the time between references is so long that memory space could probably be better utilized for a value that is referenced more frequently.

The final class of values has a short to medium interreference time and a larger standard deviation. This class corresponds to values that are referenced irregularly. We like to think that these values tend to be referenced frequently for a bit, then ignored for a long time, then referenced frequently again. If this is the case, then these values could be allocated memory only during bursts of references. The other possibility is that the values are referenced at random intervals, in which case defining a memory allocation policy would be more difficult.

In an effort to draw some more precise boundaries, statistics on various parameters histogrammed. In some cases, these statistics suggest classification by more than one parameter, so two dimensional, tabular, histograms were used. The bounds for one parameter are shown on the X axis, while the bounds for the other parameter are shown on the Y axis. The size of a class is shown numerically at the intersection of the sets of bounds.

Table 4.2 is a histogram of interreference standard deviation vs. average interreference time. The bounds for the classes were chosen to be powers of 10. Histograms are usually constructed with equal size bins, but in this case, many values are clustered near the origin, while there are few with large coordinates. Thus, greater resolution was desired near the origin. The number in each class specifies the fraction of all values that are in that class. A class with bounds A and B implies that the values in that class have a statistic on the interval $[A, B)$. A "<" in a class implies that the class contains less than 1% of the values.

The classes with interreference times between zero and 10 contain the vast majority of the variables. The classes with standard deviation 0-1 and average interreference times 0-1000 contain 62% of all values. Plotting the same histogram, but only for values that are referenced exactly

twice yields table 4.3. Notice that 44% of the values, a subset of the 62%, are referenced only twice. This group of values is a prime candidate for allocation to local memory, because of the relatively frequent referencing and short lifetimes of the values in these classes. Because these values are referenced only twice, their interreference times and their lifetimes are the same.

The class in the upper left hand corner of table 4.2 has an interreference time and standard deviation on [0,1). Furthermore, this class disappears in figure 4.3. This class contains the values that are referenced only once while the program is executing in user mode. This class includes instructions executed only once, initialized data, data input and output by the operating system, and data written once and never read.

A histogram of lifetime vs. number of references is shown in table 4.4. The classes with 1 to 10 references contain 96% of the values. Note that single-reference values are assigned lifetime 0. This histogram is especially useful for computing the bounds developed in the previous section, shown in table 4.5. For each histogram class, the top number is the upper bound on the maximum live from that class. The second number is the upper bound on the average live, and the bottom

Table 4.5. Max and average upper bounds for troff.

Lifetime	1-10	10-100	100-1000	1000-10000	10000-100000
1-10	5				
	1				
	0				
10-100	50	5			
	4				
	0				
100-1000	500	50	5		
	19	2			
	1	0			
1000-10000	5000	500	50	5	
	26	13	13		
	2	1	1		
10000-100000	50000	5000	500	50	5
	353	177	177		
	35	17	17		
100000-1000000	500000	50000	5000	500	50
	704	352	352	352	352
	352	176	176	176	176

number is the lower bound on the average live. The accuracy of three of these bounds as quick estimators is evident from figures 4.2-4.4, which show the actual number of live values from three classes as a function of time, measured in references.

The histograms up to this point have counted the number of values in each class, which offers a measure of the amount of memory required per class. However, we are also interested in the traffic, and the referencing activity caused by each class. This is shown for troff with an interreference time vs. standard deviation histogram in table 4.6. In this histogram, the numbers in each class indicate the fraction of the total references to values in that class. Comparing table 4.6 with table 4.2, one is struck by the shift in the weights of classes. Classes that contain many values may account for relatively few references, while classes with few values may account for many references.

The cause of this shift in the weights of the classes is better illustrated in table 4.7. Notice that the classes with greater reference bounds account for a larger portion of the references. Thus, while the classes with small interreference times and few references account for a large percentage of the values yet require little memory space, the classes with fewer values, longer interreference times and more references make a more significant contribution to the traffic. It is thus important that these classes be allocated space in memory.

Consider the class in table 4.7 with 100-1000 references and a lifetime of 100,000-∞. This class contains about 1% of the values, yet almost $\frac{1}{3}$ of all references are to this class. Furthermore, the average number of live values in this class, from table 4.5, is estimated as $\frac{352 + 176}{2}$=264 out of a total of approximately 352 values in the class. The long lifetimes of the values in this class result in the large number of average live.

Earlier, a distinction was mentioned between values and variables. Figure 4.5 shows a plot of interreference standard deviation vs. average interreference time for variables in the unflattened trace. Because this plot involves memory locations instead of values, as in figure 4.1, the

referencing patterns are not as uniform, as evidenced by a more spread out plot. In particular, more of the points have a large standard deviation. Each memory location contains several values, each of which may have distinct referencing behavior. The referencing patterns for values of the same variable are averaged together in figure 4.5, whereas they are displayed independently in figure 4.1. Furthermore, because the memory locations are not killed off on the last read before a write, their lifetimes are longer. Similarly, the period during which the memory location is dead, between a write and the previous read, is included in calculating the interreference time.

Table 4.8 is a histogram of interreference time standard deviation vs. average interreference time for variables. The numbers in the classes indicate the fraction of all variables in each class. Comparing this table to table 4.2, one finds that, with the exception of the class in the upper left, the classes with the smaller bounds contain a smaller percentage of the variables than do the corresponding percentage of values in table 4.2. Reasons mentioned for this shift are the same as above. Values with smaller interreference times may occupy the same memory location as values with larger interreference times. All the interreference times for a variable are averaged together, resulting in fewer small interreference times. Furthermore, the standard deviations are larger because of the more diverse referencing patterns. Notice also that the trace contains only 5,053 variables as opposed to 35,203 values. Furthermore the value trace has 342,726 references as compared to 289,517 in the value trace. The greater number of references is a result of including move instructions in the unflattened variable trace. The class in the upper left includes the variables that are referenced once, similar to those in figure 4.2.

A similar histogram, showing the fraction of references to each variable class is shown in table 4.9. Notice once again that the classes with the most variables do not necessarily correspond to the classes with the greatest impact on traffic.

Table 4.10 shows a histogram of lifetime vs. number of references for variables. The trend here as compared to table 4.4 is toward longer life times and more references per variable. The variable trace has $\frac{342726}{5053} = 67.83$ average references per variable as opposed to $\frac{289517}{35203} = 8.22$

references per value in the flattened trace. The unflattened variable trace has 18% more references, and the number of values is 7 times the number of variables. In the variable trace, a memory location is not considered dead until its final reference in the trace.

Finally, a histogram with the same axes, counting references is shown in table 4.11. Classes with less than 1% of the variables account for 19% and 32% of the references. Because these two classes, with lifetimes of 100,000–∞ and references of 1,000–10,000 and 10,000–100,000, contain so few values, their average memory space requirement should be less than 50 memory location (variables) each, yet these classes collectively account for over half of all the references.

A small assortment of data for the other three programs, diff, flt, and gauss is shown in tables 4.12–4.17. The trends described above are also apparent in these histograms.

5. Conclusions

The bounds described above, along with the results from the trace analysis system offer guidelines for determining the traffic improvements that can be obtained from a memory of a specific size during the execution of particular programs. Analyzing a sufficient number of representative programs provides statistics for use in the bounds calculations above that determine the performance estimate of the local memory under actual use. The bounds do not provide the kind of detailed performance data required for final design and tuning of a memory, but they do offer a basis for making general decisions regarding memory design.

In addition to judging design, the bounds developed in this research offer guidelines for resolving the tradeoffs involved in performing allocation of the local memory. Knowledge of the memory requirements of different classes and their effect on traffic, allows more informed decisions to be made regarding memory allocation policies, to be implemented at either the compiler or the hardware level.

Analyzing a value trace offers information about the referencing behavior of a variable only during its live periods. The live periods are the only times that the value needs to be allocated

space in local memory, thus the memory referencing behavior during these periods is significant. Live value analysis allows the memory allocation mechanism to be more finely tuned to discriminate between live variables and dead ones. Furthermore, the referencing behavior of a memory location is a composite of the referencing behavior of the values that occupy it during the execution of the program. Value Behavior, in contrast with variable behavior, is much more coherent, easier to classify, and relevant to resolving the tradeoffs at issue here. Analyzing a value trace allows studying the referencing behavior of the values separately.

Acknowledgement: This research was supported by the National Aeronautics and Space Administration (NASA) under contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), a NASA-supported Center for Excellence

REFERENCES

[Smit82]
A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, September 1982.

[Stre83]
W.D. Strecker, "Transient Behavior of Cache Memories," *ACM Transactions on Computer Systems*, vol. 1, pp. 281-293, November 1983.

[Clar83]
D.W. Clark, "Cache Performance in the VAX-11/780," *ACM Transactions on Computer Systems*, vol. 1, pp. 24-37, February 1983.

[Haik84]
I.J. Haikala, "Cache Hit Ratios with Geometric Task Switch Intervals," in *The 11th Annual Symposium on Computer Architecture*, Ann Arbor, MI, pp. 364-371, June, 1984.

[AKCB86]
C.A. Alexander, W. Keshlear, F. Cooper, and F. Briggs, "Cache Memory Performance in a UNIX Environment," *Computer Architecture News*, vol. 14, pp. 41-70, June, 1986.

[PeSh77]
Peuto and Shustek, "An Instruction Timing Model of CPU Performance," *4th Annual International Symposium on Computer Architecture*, pp. 165-178, March 1977.

[Smit85]
A.J. Smith, "Sprunt Cache Evaluation and the Impact of Workload Choice," in *The 12th Annual Symposium on Computer Architecture*, Boston MA, pp. 64-75, June, 1985.

[MGST70]
R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Sytems Journal*, vol. 9, pp. 78-117, 1970.

[Bela66]
L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal*, vol. 5, pp. 78-101, 1966.

[HKMW66]
L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, "Index Register Allocation," *Journal of the ACM*, vol. 13, pp. 43-61, January 1966.

[Day70]
W. H. E. Day, "Compiler Assignment of Data Items to Registers," *IBM Systems Journal*, vol. 14, pp. 281-317, 1970.

[Beat74]
J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM Journal of Research and Development*, vol. 18, pp. 20-39, January 1974.

[Chai82]
G.L. Chaitin, "Register Allocation and Spilling via Graph Coloring," in *SIGPLAN 82 Symposium on Compiler Construction*, Boston MA, pp. 98-105, June 23-25, 1982.

Table 4.2. Average Interreference Time vs. Standard Deviation
Program troff, Flattened trace, 289517 references
Fraction of35203 values

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.18	0.29	0.06	0.09	0.02	0.03	<
1-10		0.18	0.02	<	<	<	
10-100		<	0.06	0.02	<	<	
100-1000			<	0.03	<		
1000-10000				<	0.02		
10000-100000					<	<	
100000-∞							

Table 4.3. Average Interreference Time vs. Standard Deviation
Program troff, Flattened trace, 289517 references, References = 2
Fraction of 35203 values

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.29	0.06	0.09	<	<	<
1-10					<	<	
10-100						<	
100-1000							
1000-10000							
10000-100000							
100000-∞							

Table 4.4. Number of References vs. Lifetime
Program troff, Flattened trace, 289517 references
Fraction of 35203 values

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.18					
1-10		0.29					
10-100		0.28					
100-1000		0.15	<				
1000-10000		0.02	<	<			
10000-100000		0.02	0.01	<			
100000-∞		0.02	0.01	0.01	<	<	

Table 4.6. Average Interreference Time vs. Standard Deviation
Program troff, Flattened trace, 35203 values
Fraction of 289517 references

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.29	0.06	0.09	<	<	<
1-10					<	<	
10-100						<	
100-1000							
1000-10000							
10000-100000							
100000-∞							

Table 4.7. Number of References vs. Lifetime
Program troff, Flattened trace, 35203 values
Fraction of 289517 references

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.02					
1-10		0.07					
10-100		0.13					
100-1000		0.05	<				
1000-10000		<	0.02	0.02			
10000-100000		<	0.04	0.02			
100000-∞		0.01	0.04	0.30	0.13	0.11	

Figure 4.8. Average Interreference Time vs. Standard Deviation
Program troff, Unflattened trace, 342726 references
Fraction of 5053 variables

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.34	<	0.01	0.02	0.08	0.27	0.02
1-10		<	<	<	0.05	<	
10-100			<	0.02	<	<	
100-1000			<	0.03	<		
1000-10000				0.04	0.08	<	
10000-100000					<	<	
100000-∞							

Figure 4.9. Average Interreference Time vs. Standard Deviation
Program troff, Unflattened trace, 5053 values
Fraction of 342726 references

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	<	<	0.01	0.10	0.08	0.03	<
1-10		0.11	<	<	<	<	
10-100			0.13	<	<	<	
100-1000			0.23	0.04	<		
1000-10000				0.19	0.06	<	
10000-100000					<	<	
100000-∞							

Table 4.10. Number of References vs. Lifetime
Program troff, Unflattened Trace, 342726 references
Fraction of 5053 values

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.34					
1-10		<					
10-100		<					
100-1000		0.01	<				
1000-10000		0.01	<	<			
10000-100000		0.15	0.07	<			
100000-∞		0.14	0.14	0.10	<	<	

Table 4.11. Number of References vs. Lifetime
Program troff, Unflattened trace, 5053 values
Fraction of 342726 references

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		<					
1-10		<					
10-100		<					
100-1000		<	<				
1000-10000		<	<	<			
10000-100000		<	0.02	<			
100000-∞		0.01	0.06	0.36	0.19	0.32	

Table 4.12. Average Interreference Time vs. Standard Deviation
Program diff, Flattened trace, 43807 values
Fraction of 254020 references

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.04	0.07	0.11	<	<	0.01	<
1-10		0.03	0.02	<	<	<	
10-100			0.16	<		<	
100-1000			0.29	0.17	<		
1000-10000				0.03	0.05		
10000-100000					<	<	
100000-∞							

Table 4.13. Number of References vs. Lifetime
Program diff, Flattened trace, 43807 values
Fraction of 254020 references

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.04					
1-10		0.07					
10-100		0.19	<				
100-1000		0.02	<	<			
1000-10000		<	<	<			
10000-100000		0.02	<		0.27		
100000-∞		<	0.03	0.10	0.22		

Table 4.14. Average Interreference Time vs. Standard Deviation
Program fit, Unflattened trace, 8223 values
Fraction of 339698 references

Standard Deviation	Average Interreference Time						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.02	<	<	<	0.07	0.02	<
1-10			<			<	
10-100			0.33				
100-1000			0.20	0.24	0.03	<	
1000-10000				0.02	0.07		
10000-100000						<	
100000-∞							

Table 4.15. Number of References vs. Lifetime
Program fit, Unflattened trace, 8223 values
Fraction of 339698 references

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.02					
1-10		<					
10-100		<					
100-1000		<					
1000-10000		<					
10000-100000		<	<				
100000-∞		<	0.08	0.22	0.42	0.26	

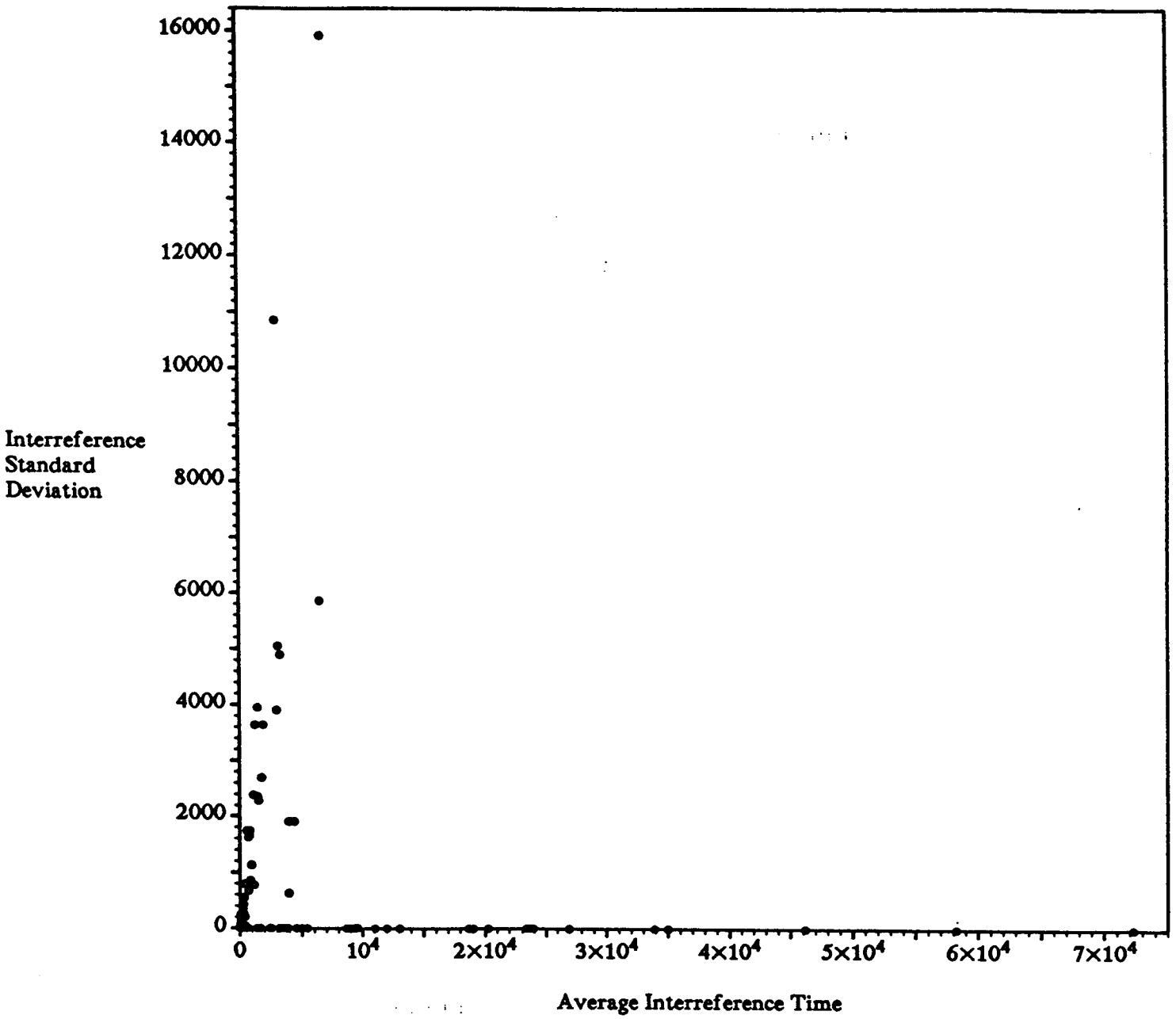
Table 4.16. Number of References vs. Lifetime
Program gauss40, Flattened trace, 242870 references
Fraction of 31029 values

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.18					
1-10		0.23					
10-100		0.39	0.01				
100-1000		0.10	<				
1000-10000		0.01	<				
10000-100000		<	<				
100000-∞		0.06	<	<	<	<	

Table 4.17. Number of References vs. Lifetime
Program gauss40, Flattened trace, 31029 values
Fraction of 242870 references

Lifetime	Number of References						
	0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.02					
1-10		0.08					
10-100		0.12	0.01				
100-1000		0.03	0.04				
1000-10000		<	<				
10000-100000		<	<				
100000-∞		0.02	0.06	0.16	0.31	0.11	

Figure 4.1. I-ref Standard Deviation vs.
Average I-ref Time
Program troff, Flattened Trace
289517 references, 31853 values,1000 points Plotted



4.2

fig 8.9

Life ∈ [100, 1000)

Pfs ∈ [1, 10

Live value count

Program trace

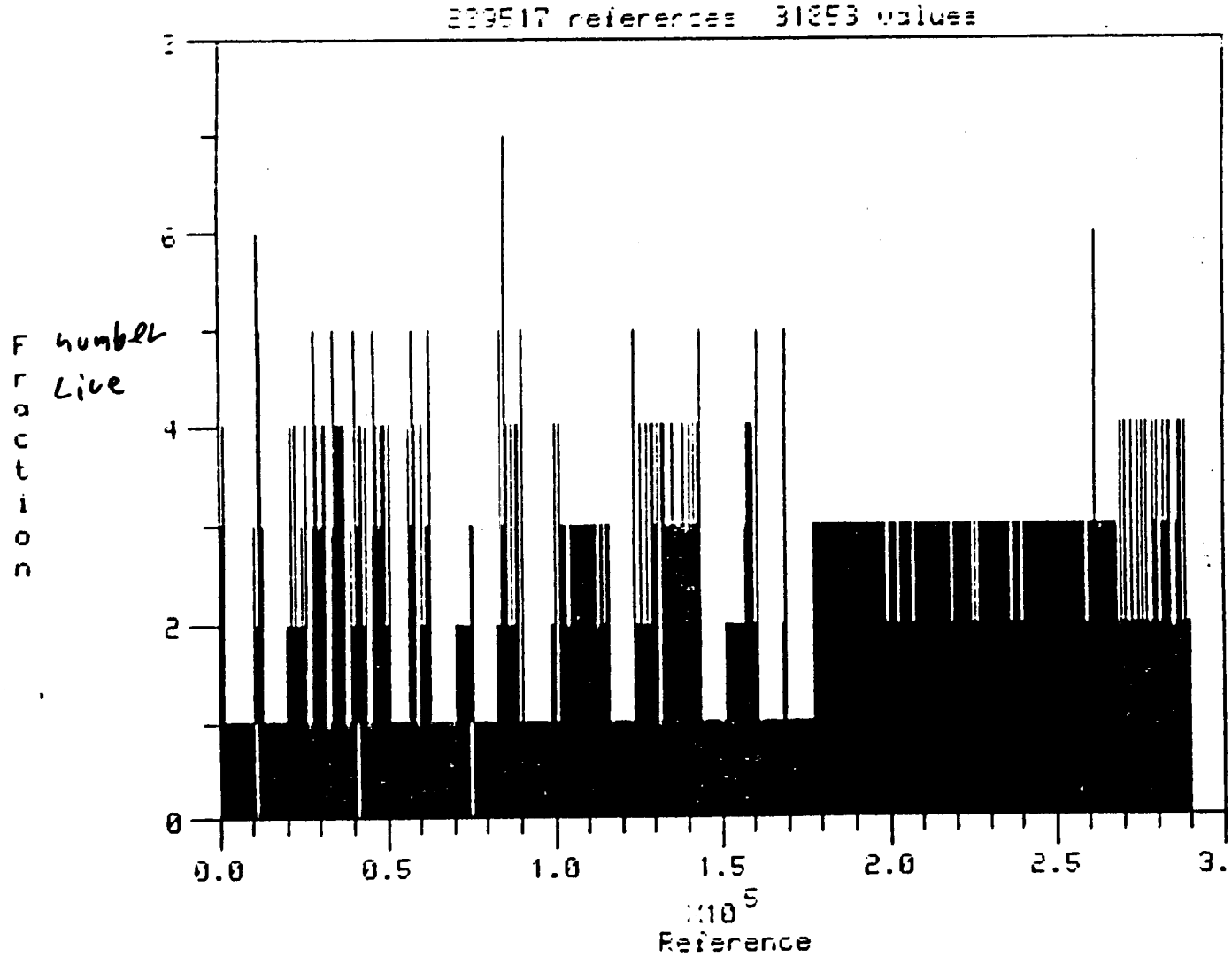


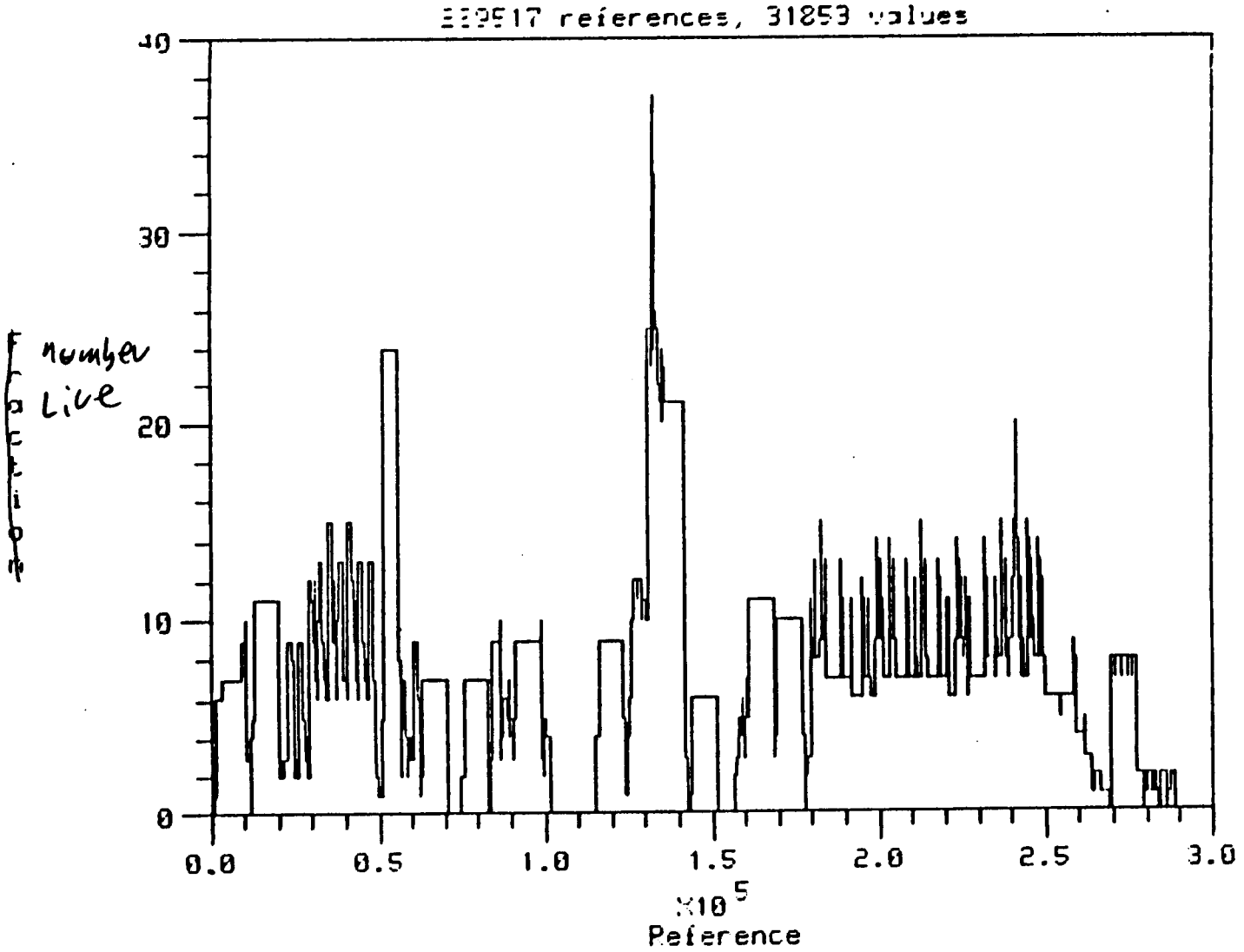
fig 4.3

life $\in$ [1,000, 10,000)

refs $\in$ [7, 10)

~~Live value count~~ Live value count

~~Program trace~~ Program trace



life ∈ [100,000 284,518)

refs ∈ [1, 10)

fig 4.4

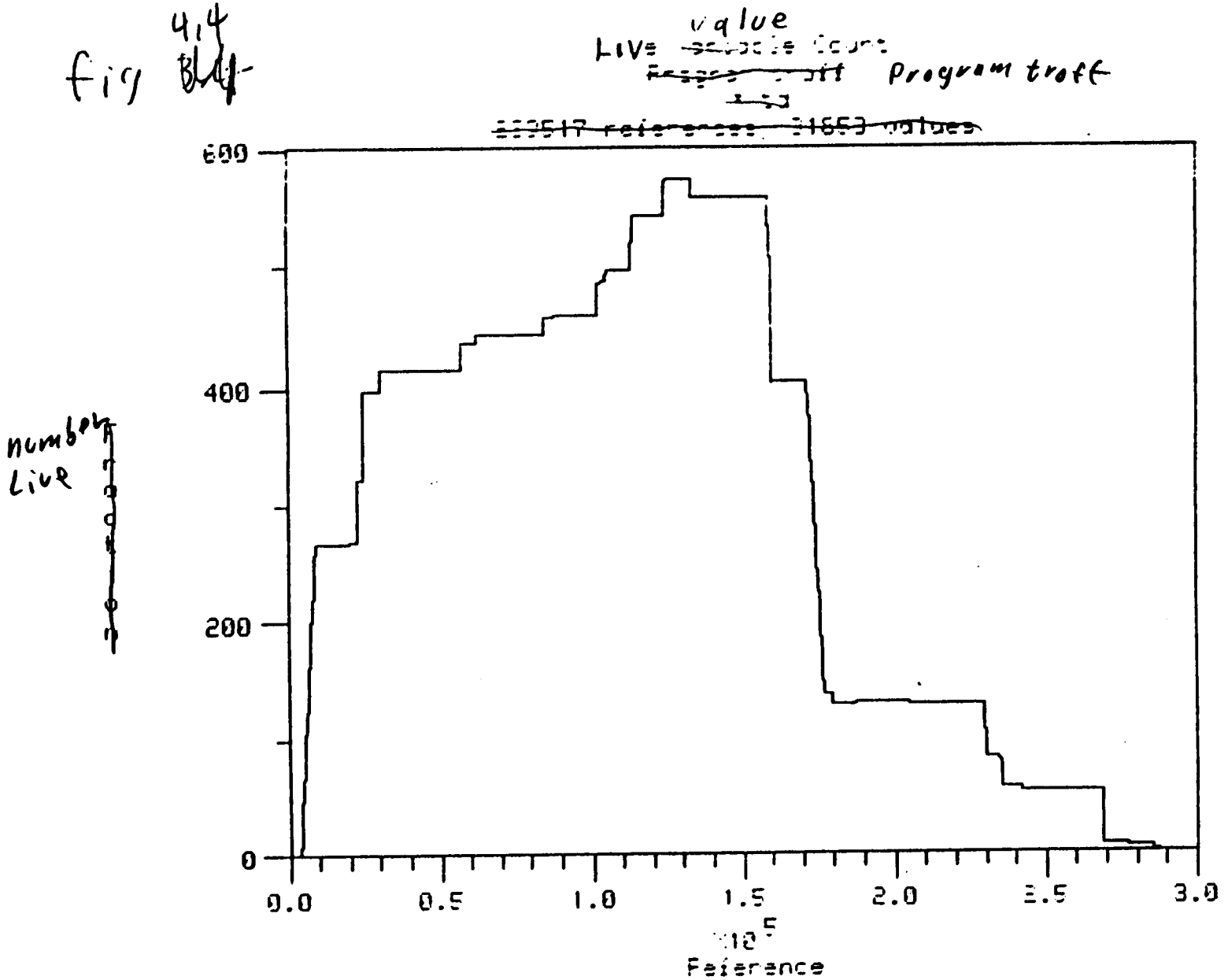


Figure 4.5. I-ref Standard Deviation vs.
Average I-ref Time
Program troff, Unflattened Trace
342716 references, 5053 variables, 1000 points plotted